System Verification and Validation Plan for COMPSCI 4ZP6

RatBAT: Rat Behavioral Analysis Tool

Team 8

Name	Email
Brandon Carrasco	carrascb@mcmaster.ca
Daniel Locke	locked3@mcmaster.ca
Jamie Wong	wongj171@mcmaster.ca
Inoday Yadav	yadavi1@mcmaster.ca

Table of Contents

1 Revision History 3
2 Project Description 4
SRS Document: Link, Design Document: Link 4
3 Component Test Plan 4
3.1 Summary Measure Interface 4
3.1.1 Testing Methodology 4
3.1.2 Performance Tests & Metrics 4
3.2 Summary Measure Package 5
3.2.1 Testing Methodology 5
3.2.2 Performance Tests & Metrics 5
3.3 UI/Frontend 6
3.3.1 Testing Methodology 6
3.3.2 Performance Tests & Metrics 6
3.4 Backend 7
3.4.1 Testing Methodology 7
3.4.2 Performance tests and metrics 7
3.5 Web Platform April 2025 Update (Data Prepreprocessing, Compute Summary Measures, Compile Data pages) 8
3.5.1 Testing Methodology 8
3.6 Data Storage & Importing 8
3.6.1 Testing Methodology 8
3.6.2 Performance Tests and Metrics 8
3.7 Data Preprocessing 8
3.7.1 Test Methodology 8
3.7.2 Performance Tests & Metrics 9

1 Revision History

Date	Version	Notes
1/24/2025	0	Original draft of document
04/03/2025	1	Final version of document

2 Project Description

This project aims to leverage and expand upon the existing dataset generated from the Quinpirole Sensitization Rat Model of OCD experiments, which involves tracking rat movements in an open field and collecting corresponding x, y, t positional data. This experiment has been running for multiple years, with the majority of the data being stored in the FRDR. Though there exists a substantial amount of raw data, there is currently no easy and straightforward method to access, process, and analyse this data. Our project focus is to develop a robust and accessible open-platform web application for researchers. This application will facilitate the preprocessing, computation of summary measures, analysis, and collection of rodent behavioural data from the FRDR repository.

SRS Document: Link, Design Document: Link

3 Component Test Plan

3.1 Summary Measure Interface

3.1.1 Testing Methodology

PyTest will be the primary testing framework for assessing the correctness and performance of the summary measure interface. The correctness will be assessed using assertions that ensure that all required summary measures and common calculations are added to a request.

- Correctness Procedure
    - Karpov
        - **Assert** that all summary measures' dependencies are within list of summary measures to be calculated post-processing
        - **Assert** that all common calculations are within the list outputted by Karpov after post-processing.
- **RESULTS**
    - All tests are passing for all summary measures.

3.1.2 Performance Tests & Metrics

Performance tests will cover the end-to-end travel time of a full request, saving these metrics in separate files for automatic future reference when optimizing or altering components of the Interface. Metrics will mainly target time to process requests, to calculate common calculations, to calculate summary measures, and the full run-through time (from end-to-end).

- Performance Procedure
    - **Run** end-to-end requests, recording the following metrics into a separate file:
        - Time to process a request using Karpov
        - Time to calculate all common calculations
        - Time to calculate all summary measures

        - Overall run-through time (from request sent to results returned).
    - **Compare** new performance time with previous performance time and highlight any significant improvements or declines.
- **RESULTS**
    - Comparison files haven't been created, however calculations are being performed within **O(n^2**) time, or less than one second.

3.2 Summary Measure Package

3.2.1 Testing Methodology

PyTest will be the primary testing framework for testing the truthfulness and performance of all defined summary measures. For truthfulness — assertions that cover that all functions output correct values — our supervisors have provided several files that have ground truth values (outputs of summary measures) associated with many different experiments (or data) files. Through comparison with those ground truth values, if any summary measure algorithm is incorrectly defined, it'll be caught and corrected.

- Truthfulness Procedure
    - Summary Measures
        - **Assert** that summary measures (given data from an experiment) will return the same or similar (<= 1 difference) values as the ground truth calculations of that summary variable (from the same experiment).
- **RESULTS**
    - Majority of tests are failing due to a difference in methodology of how stops (lingering episodes) are calculated. Although we're using the supervisor's current definition of stops, this may not be how the summary measures (and their stops) were originally calculated. Further investigation by ourselves and supervisors are required to resolve this issue.
        - Should stops be calculated in accordance to their original methodology, all tests are expected to pass.

3.2.2 Performance Tests & Metrics

PyTest will also provide the capability to ensure that algorithms do not violate the polynomial time requirement, as well as provide performance metrics for algorithm optimization. The performance metrics will be recorded and saved in a separate file for the developer's reference. Automatic comparisons to these saved metrics shall inform the developers if any alterations to an algorithm is better optimized or not.

- Performance Procedure
    - **Run** all summary measures and record the following metrics.
        - Time to calculate summary measure

    - **Compare** new performance time with previous performance time and highlight any significant improvements or declines.
- **RESULTS**
    - All tests are currently passing < 1, with each summary measure averaging 0.6 seconds.

Metrics	Expected Performance
Algorithm Runtime	Polynomial Time

3.3 UI/Frontend

3.3.1 Testing Methodology

We will use Jest as the primary unit testing framework for the frontend to test our React components. We'll write unit tests for each component to ensure they render and function correctly, such as handling input, displaying data, and allowing interactions like data filtering and visualization. Additionally, we will test edge cases such as the user attempting to apply changes without selecting a file which gives an appropriate alert popup message. For static code analysis, we use ESLint for React frameworks to check for syntax errors, best practices, and identify unused variables or unreachable code. Additionally, Black Box Testing will be used to check web interactions and behaviors. For example, do filter combinations in our database query pull the correct corresponding data?

3.3.2 Performance Tests & Metrics

For performance testing, we will measure the latency of API responses when fetching data files and summary measures using Postman. We will also use Lighthouse to measure page rendering speed and user interaction responsiveness to hopefully identify bottlenecks affecting UI performance. Our target is to ensure API calls are complete within 500ms under normal load conditions. We're aiming for page rendering and UI responsiveness to be 100-200 ms response time.

Metrics	Expected Performance
Page Rendering	< 1 seconds
UI elements responsiveness	< 200ms
Query-data Response Time	< 500ms
JSON/CSV File Generation	< 3 seconds

Data Processing Filtering	< 2 seconds

3.4 Backend

3.4.1 Testing Methodology

We will use PyTest along with pytest-django as the primary testing framework for the backend. PyTest allows us to create isolated tests for each function, ensuring that individual components behave as expected, especially when handling large datasets or sensitive behavioral data. Pytest-django allows integration tests to run smoothly, particularly for database interactions, API endpoints, and data workflows. We will also utilize cURL to verify API endpoints directly throughout development, ex.

```
curl -X POST http://127.0.0.1:8000/api/data-preprocessing/preprocess/ \
-H "Content-Type: application/json" \
-d '{"selectedTrials": ["33"], "parameters": { "LOWESS": {"deg": 2, "half_window": 24, "num_iter": 2}, "RRM": {"half_windows": [7, 5, 3, 3], "min_arr": 12, "tol": 1.3}, "EM": {"tol": 0.000001, "half_window": 4, "log_transform": "cbrt", "num_guesses": 5, "num_iters": 200}}, "determineKAutomatically": true}'
```

3.4.2 Performance tests and metrics

Performance testing will focus on measuring the time taken to compute summary measures based on different file sizes and hardware resources. Apache JMeter will be used to simulate multiple users to see system scalability and evaluate database queries during high load.

Metrics	Expected Performance
High Load API Requests	< 1 minute for 95% of requests
50 Concurrent API Requests	No Query Deadlocks
Memory Usage	< 5 GB peak memory
CPU Utilization	< 50% sustained load

3.5 Web Platform April 2025 Update (Data Prepreprocessing, Compute Summary Measures, Compile Data pages)

3.5.1 Testing Methodology

From file loading across pages being implemented and the last three pages of the web platform being very dependent on each other now, following 3.3 and 3.4 testing methodology will no longer be possible for these pages due to time and feasibility constraints. To tackle this we will be creating a manual testing plan which addresses all primary functionality of the Data Prepreprocessing, Compute Summary Measures, and Compile Data pages. It will be a step-by-step procedure that the tester can follow to quickly ensure end-to-end correctness of the web platform from loading trial data from the database, to preprocessing it, computing summary measures on it, and finally compiling all data into downloadable files.

3.6 Data Storage & Importing

3.6.1 Testing Methodology

Primary testing for the database and data import and structuring functionality will be through PyTest and pytest-django. This will allow us to do unit testing on all individual data storage functions and allow us to test edge cases for each function. For the sake of unit testing we will be using a test database in place of our production database to ensure that our testing data does not linger in our actual applications, this can be easily achieved through Django's testing framework.

3.6.2 Performance Tests and Metrics

The major focus of this testing will be on the management of high data loads and ensuring that importing large volumes of data all at once will not cause our system to fail. For instance, through the FRDR Query module, we will supply wide ranging filters that result in large volumes of data being downloaded and stored into our database at once and ensuring that we do not see any unintended behavior as a result.

3.7 Data Preprocessing

3.7.1 Test Methodology

Again, PyTest will be used for testing the correctness and performance of data preprocessing. Data preprocessing is divided into two subcomponents: smoothing and segmentation. The correctness of smoothing will be tested by comparing the results of smoothing to pre-existing ground truth data points of smoothed data. As there might be some slight variation in data due to some floating point arithmetic, we will allow some difference between the two within some

specified epsilon value. For further robustness, our unit testing will also include testing against other smoothing algorithms whose accuracy as compared to the algorithm we are implementing (SPSM) is known for specific cases.

3.7.2 Performance Tests & Metrics

For testing the accuracy of our smoothing subcomponent (an implementation of the SPSM smoothing algorithm), as we are unable to actually test how closely our smoothed data matches with actual rodent behavior in the trials, we will implement an alternative smoothing algorithm (Moving Average Smoothing (MA)) and test our algorithm against our reference algorithm in its ability to accurately measure total distance traveled over segments of rodent inactivity (movement caused my noise rather than actual rodent movement), and total distance traveled in segments of high rodent activity. As demonstrated in *Journal of Neuroscience Methods 133 (2004) 161-172* the following performance metrics will ensure the accuracy of our algorithm:

Metric	Expected Performance
Distance - Segments of Rodent Inactivity	**total_distance(SPSM)** < **total_distance(MA)**
Distance - Segments of High Rodent Activity	**total_distance(SPSM)** > **total_distance(MA)**

Performance testing will also cover the segmentation subcomponent (the classification of data intervals into lingering and progressing segments). Several Machine Learning metrics apply here, such as accuracy, recall, precision, and F1-score. These metrics will be recorded and automatically compared for the developer's reference. The 'true' class labels for this testing is smoothed data that we have for a subset of the database produced using a different implementation of the segmentation algorithm that we do not have access to. The data we have to test against all uses a specific set of parameters so we will test the performance of our algorithm against this data using these specific parameters and use that information to determine the correctness of our algorithm for any provided parameters.

- Performance Procedure - Segmentation
    - **Record** all of the following metrics:
        - Accuracy
        - Recall
        - Precision
        - F1-score

Metric	Expected Performance

Accuracy	> 95%
Recall	> 90%
Precision	> 90%
F1-score	> 90%

4 Component Test Results

4.3 Frontend (FRDRQuery Component)

The FRDRQuery page underwent both automated testing and manual validation to ensure reliable functionality and alignment with stakeholder expectations.We implemented unit tests to verify the core functionality of all major components on the page. All tests passed successfully:

DataWindow.test.jsx - PASSED
FieldSelector.test.jsx - PASSED
FilterButton.test.jsx - PASSED
FRDRQuery.test.jsx - PASSED

These tests covered:

- Component rendering
- User interactions (e.g. clicking buttons, applying filters)
- Integration between filters and data fetching
- Simulated API responses using test doubles (mocks)

Note! These tests focused primarily on filtering by trial_id. Other filters were validated manually due to time constraints and test complexity.
Though these test for proper functionality of the components and the FRDRQuery page. However, true api calls and all filtering options were not covered (we mainly used trial_id to filter). This was due to time constraints and infeasibility.

To ensure all filtes worked as intended, we conducted manual testing on each available filter using knowledge of expected ranged provided by our supervisors. Testing included:

- Exact matches
- Range inputs
- Numeric comparisons (>, <, <=, >=
- Text lookups (contains, endswith, etc.)
- Discrete checks

Outcome: No issues were identified, All filters returned expected results

While our unit tests used mocked API responses, we validated real API interactions through the web interface:

Filter Input	Action	Expected Output	Outcome
Trial_id <= 10	Load Data from FRDR	All files loaded successfully!	PASSED
Trial_id <= 100	Load Data from FRDR	All files loaded successfully!	PASSED
trial_id: exact 1	Download Timeseries CSV	ZIP file contains all expected trials	PASSED
trial_id: range min 0 max 10	Download Timeseries CSV	ZIP file contains all expected trials	PASSED
trial_id: <= 10	Download Timeseries CSV	ZIP file contains all expected trials	PASSED

Note! Sometimes partial and full errors were observed. These were traced to external issues with FRDR, which is a independent platform (Out of our control).

Metrics	Expected Performance (Avg)
Page Rendering	300 ms
UI elements responsiveness	290ms
Query-data Response Time	~4 minutes
JSON/CSV File Generation	~8 minutes

Performance Analysis:
Though the FRDRQuery page performed all intended functions correctly, its overall performance was lower than anticipated due to significant delays caused by external dependencies. The primary bottleneck was the frdr-query API call, which, under heavy load from the FRDR backend, could take several minutes to complete. This was particularly evident when querying large datasets involving numerous trials. Additionally, the get-timeseries endpoint introduced further delays, as it required retrieving and zipping hundreds of large CSV files, each containing extensive time-series data. These operations placed considerable strain on both the FRDR server and the client-side application, resulting in prolonged load times. While the user interface handled these delays appropriately through loading indicators and alerts, the responsiveness of

the platform was ultimately constrained by the performance limitations of the FRDR infrastructure, which are beyond the control of this system. Performance was original expected to be significantly higher based on our initial plans with a Globus subscription. However, due to limitations in McMaster's subscriptions, we were unable to access this option.

4.7 Data Preprocessing

For tests on both the smoothing and segmentation algorithms, the algorithms were run on approximately 2 million data points to produce the below results.

4.7.1 Smoothing

Test for accuracy in periods of high rodent activity (the SPSM algorithm uses LOWESS for smoothing in these periods):

- Total distance travelled for the SPSM algorithm (LOWESS): 281505.19 cm
- Total distance travelled for the MA algorithm: 223932.89 cm
- Performance metric: total_distance(SPSM) < total_distance(MA)
- Test result: **PASSED**

Test for accuracy in periods of low rodent activity (the SPSM algorithm uses RRM to identify these periods):

- Total distance travelled for the SPSM algorithm (RRM): 97982.54
- Total distance travelled for the MA algorithm: 116270.80
- Performance metric: total_distance(SPSM) > total_distance(MA)
- Test result: **PASSED**

4.7.2 Segmentation

Resulting Performance:

- Accuracy: 96.19%
- Precision: 74.91%
- Recall: 67.57%
- F1 score: 71.05%

Performance Analysis:

The primary reason for lower than anticipated performance on precision, recall and F1 score is probably due to a flaw that exists in the original algorithm that we were implementing described in *Journal of Neuroscience Methods 96 (2000) 119-131* is also the case for our algorithm. The flaw is that the algorithm will often miscalculate the threshold for segmentation due to inaccurately high peaks in the lingering movement distribution, as a result of this, when the original data was produced that we are testing against, manual changes were made to many of the segmentation thresholds meaning the automatically determined thresholds were not used.

Since we are testing using automatic thresholds, this leads to a notable level of error caused by the fact that the automatic thresholds were not used for much of our ‘true’ labels.

As well the segmentation algorithm introduces a level of randomness in its implementation. This comes in the form of the initial guesses made by the algorithm about what parameters to use to model the movement distribution. This randomness means we cannot exactly replicate conditions that produced the values we are testing against.